

DINAMITE: A modern approach to memory performance profiling

Svetozar Miucin

Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
smiucin@ece.ubc.ca

Conor Brady

School of Computing Science
Simon Fraser University
Burnaby, Canada
cbrady@sfu.ca

Alexandra Fedorova

Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
sasha@ece.ubc.ca

Abstract—Diagnosing and fixing performance problems on multicore machines with deep memory hierarchies is extremely challenging. Certain problems are best addressed when we can analyze the entire trace of program execution, e.g., every memory access. Unfortunately such detailed execution logs are very large and cannot be analyzed by direct inspection. We present DINAMITE: a toolkit for Dynamic INstrumentation and Analysis for MassIve Trace Exploration. DINAMITE is a collection of tools for end-to-end performance analysis: from the LLVM compiler pass that instruments the program to plug-and-play tools that use a modern data analytics engine Spark Streaming for trace introspection. Using DINAMITE we found opportunities to improve data layout in several applications that resulted in 15-20% performance improvements and found a shared-variable bottleneck in a popular key-value store, whose elimination improved performance by 20x.

Index Terms—instrumentation, memory optimizations, LLVM, Spark Streaming

I. INTRODUCTION

Memory performance is a limiting factor in many important programs. Traditional database systems, web servers, scientific algorithms and modern data analytics programs alike were observed to spend 50-80% of CPU cycles stalled on memory [1] [2]. That is, 50-80% of the time these programs are unable to commit any instructions due to outstanding long-latency memory accesses. Understanding and addressing the causes of these bottlenecks is of paramount importance. Performance improvements from a more efficient memory layout or improved locality of access are usually significant and in some cases reach an order of magnitude [3] [4] [5] [6] [7] [8].

At the same time, optimizing memory performance is notoriously difficult. Compiler optimizations can be effective when static analysis is sufficient to infer improvement opportunities. However, the scope of static optimizations is limited [9], partly because insight into a bottleneck can often be gained only during execution and partly because the compiler is limited in how it can change data structure layout, particularly with dynamically allocated data structures and in unmanaged languages. As a result, developers often resort to manually optimizing their data structures and algorithms, relying on tools for dynamic program analysis and memory profiling.

Unfortunately, most existing tools suffer from either lack of generality, portability, or flexibility. Conventional CPU

profilers, such as perf [10], aim to identify source locations that generate the majority of cache misses, but because of skid effects in hardware counters [11], [12] or compiler optimizations, such as function inlining, this information is often imprecise or plain wrong. Cachegrind [13] accurately identifies source lines generating cache misses, but does not provide actionable insight that might lead the programmer to reduce them. Dprof [5] identifies data structures and fields that are responsible for cache misses due to sharing among threads, but it is not flexible enough to address other causes of poor memory performance and was designed specifically for the Linux kernel. Similarly, Memprof [4] focuses on identifying objects that cause remote accesses on NUMA systems, but the implementation is Linux-specific, tied to AMD hardware and does not lend itself to other types of analyses.

To address this gap, we built DINAMITE – a toolkit for Dynamic INstrumentation and Analysis for MassIve Trace Exploration. DINAMITE uses compile-time instrumentation to inject tracing code into the program. At runtime, the program generates precise traces containing every memory access, its source location and the corresponding variable name, type and value. These traces are then used to perform various memory-related analyses, for example, identifying highest cache-miss offenders, locating hot and cold fields of a data structure, correlating locality of accesses with values of variables, detecting true and false sharing, building arbitrary models of memory access patterns, and many others.

The approach of using instrumentation and tracing is by itself not new; it is used in Pin [14], Valgrind/Cachegrind [13] and other similar tools. Its main downside is high runtime overhead and very large execution traces, which can reach hundreds of gigabytes even for small programs. However, for the very challenging task of memory performance debugging this approach is often the only practical option, because certain analyses, e.g., those relying on cache simulation, can be performed only with a precise execution trace.

Key contributions of DINAMITE are as follows:

- The instrumentation is implemented as a pass in LLVM [15], so it is applicable to any language with an LLVM front-end.
- Since the instrumentation is done at compile-time, the source-level debug information assigned to trace entries

is precise and easy to extract.

- Traces are generated in binary format and with buffering, which is the most efficient method known to us. As a result, DINAMITE’s runtime overhead is similar or smaller than state-of-the-art instrumentation tools, like Pin and Valgrind.
- DINAMITE gives the user flexibility in how to handle execution traces. The traces can be stored in the file system, but if the user does not wish or cannot store these typically large traces, they can be analyzed on-the-fly using a streaming analytics engine like Spark Streaming [16] (or any other similar engine).
- DINAMITE is easy to extend with additional analysis tools. A developer can write a new tool with a few lines of Scala (if using DINAMITE with Spark) or any other language of choice. We target advanced developers who understand how software interacts with memory hierarchies of modern processors, so we wanted to give them ultimate flexibility in analysing memory traces.

We built three tools on top of DINAMITE. The first one produces variable names and source lines responsible for the highest number of cache accesses and misses. Using it, we reduced the last-level cache (LLC) miss rate of *429.mcf* from SPEC2006 by 55% and improved its performance by 12%. We also reduced the LLC missrate and improved performance of PARSEC’s *fluidanimate* by 50% and 15% respectively.

The second tool implements Chilimbi’s structure splitting algorithm [17]. Thanks to it, we reduced the LLC miss rate of SPEC2006’s *429.mcf* by 60%, with the corresponding 20% reduction in runtime.

The third tool detects program variables that are heavily shared by many threads. This tool enabled us to detect a previously known performance bottleneck in WiredTiger, MongoDB’s back-end key/value store [18] [19] [20]. Even though the bottleneck was already known and fixed before we created DINAMITE (in fact, this was one of the motivating reasons for DINAMITE), the original discovery took several weeks, while DINAMITE pin-pointed it in a few hours. Performance improvement of the read-only sequential LevelDB benchmark implemented over WiredTiger reached a factor of 20 for 32 threads.

The rest of the paper is organized as follows. Section II provides an overview of DINAMITE design. Sections II-B, II-A and II-C contain a detailed discussion of the log format, LLVM instrumentation pass and logging library. Section II-D discusses two implementations of analysis frameworks – one in native C++ and another one using Spark Streaming. Section III describes the tools we created with DINAMITE and evaluates them on three applications. Section IV discusses related work. Section V elaborates on possible avenues for future work.

II. SYSTEM DESIGN

Our system is built from three components: an LLVM instrumentation pass, a collection of output logging libraries and an analysis toolkit. A system overview is shown in

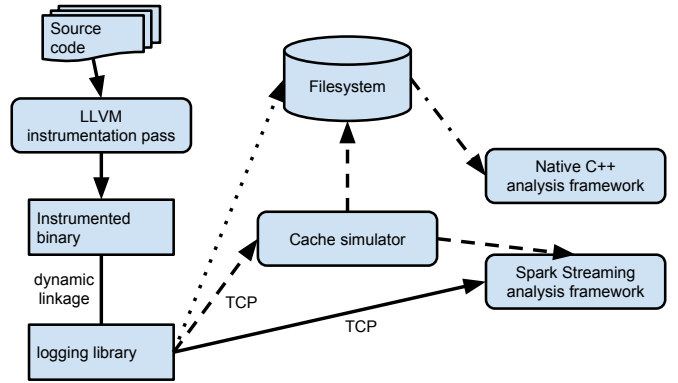


Fig. 1: DINAMITE system diagram

Figure 1. Different line types leaving the logging library show possible data paths within the system. A path taken by data depends on the type of analysis desired.

Target applications are compiled with an LLVM [15] compiler configured to include our instrumentation pass. Configuration is trivial: it only requires changing the compiler invocation command. Most large software projects allow specifying the compiler command via an environment variable.

Our instrumentation pass instruments three types of events: function entry or exit, memory allocation, and memory access. For each event the instrumentation pass injects a function call to an externally linked logging library. The logging library, linked dynamically at runtime, will produce a log record of an event in binary or text format (described below). Log records are either stored in the file system or streamed over a socket. Stored traces can be analyzed using pre-packaged DINAMITE scripts written in Python or C++ (see Section III), or the user can write her own tools using their language of choice. Log records streamed over the socket are processed by the Spark Streaming engine. DINAMITE includes several analysis kernels for Spark written in Scala; users can also write their own.

DINAMITE allows chaining analysis passes, similar to the Unix pipe command. For many of our analyses we stream the log data to a cache simulator tool (written in C++) that annotates each memory access log entry to indicate whether the access was a cache hit or a miss and forwards the annotated log entry to the Spark Streaming engine.

The rest of this section describes the system in more detail.

A. LLVM instrumentation pass

We chose to use the LLVM infrastructure for the implementation for the following reasons:

- 1) It can be used on programs written in any language supported by an LLVM front-end compiler. To date, those include, but are not limited to, C, C++, D, Haskell, Objective-C, Swift, Ruby, etc. There is even a compiler that translates Java bytecode into the LLVM intermediate representation. Given the popularity of LLVM, we can expect this list to grow in the future.
- 2) It lets us add instrumentation at the level of the intermediate representation (IR), which is more convenient

than instrumenting a binary. LLVM IR is an assembly-like language that is more abstract than machine code (e.g., it assumes an unlimited set of registers). When IR is translated into binary, a single memory access can be expanded into multiple machine instructions, which can introduce noise into traces and make it difficult to attribute accesses to source code level constructs.

- 3) Full debug information is available at IR level. The front end we used, *clang*, embeds debug information into the IR as an abstraction of the DWARF format that is easy to parse with the tools provided in the LLVM framework.

Our instrumentation pass begins by crawling the IR debug metadata to extract information about complex data types (structs, classes, unions) and categorizes them by connecting their corresponding internal LLVM references with type and field names. This is necessary because of type aliasing. A single type in C or C++ can have multiple names because of `typedefs`. Without this metadata extraction, LLVM only knows about the original type definitions, but IR instructions may contain references to different names for the same type. We store all the type alias information in map-like data structures within the pass.

The core of the instrumentation pass iterates over the current module and visits each function, each basic block and each instruction within it.

For each encountered function, it places a *function begin* log call at the beginning of its first basic block, and a *function end* log call at the end of each basic block that ends the function.

Memory allocation functions are treated separately. Our pass must first recognize whether a function is a memory allocator and then gather the information about the allocated type, size and address. For each called function, our instrumentation checks the function’s name against the list of known allocator functions. We generalize allocator functions as functions that take two arguments: *number of elements* and *size of a single element*, and output the address that points to the start of the allocated region. This model encompasses all allocation library APIs that we have encountered, and, combined with type information available through LLVM, contains all the relevant information that describes an allocation.

The list of allocators is provided in a separate file, where each entry is described with a function name, and three indices indicating the position of all the relevant fields (the number of elements, the size of the element and the allocation’s base pointer) in the argument list. If the allocation address is the function’s return value, its index will be set to `-1`. Standard allocation functions (such as `malloc` and `calloc`) are included in the configuration file that comes with our pass. If the program uses any non-standard allocator functions, the user must add them to that file. The pass places an allocation event log call after each call to an allocator function.

Strings written to the log are encoded with a unique integer identifier to preserve space. The integer-to-string mappings are placed into JSON documents created by the instrumentation pass. The mappings must be consistent across modules, but

LLVM compiler passes do not preserve any state across modules. Therefore, we load the JSON mappings before compiling each module compilation and write back any updates at the end.

For ease of use with C and C++ projects, our instrumentation pass gets registered with LLVM’s pass manager for standalone clang invocations. As clang supports most of GCC’s compilation flags, this makes integration of our instrumentation into existing projects in most cases as easy as changing the compiler invocation variable.

B. Log format

All log events contain a field for a thread identifier. We limit this field to 8 bits to conserve space, but it can be easily expanded if needed. Distinguishing between 256 unique threads was sufficient for our case studies.

Allocation and access events share fields that contain the file name, the line number and the column number that correspond to the event’s source code location. Allocation events additionally contain the base address of the allocated memory region, the size of a single allocated element, the number of allocated elements and their type.

Access events contain the accessed address, the type of the access (read or write), the name and type of the variable corresponding to the access, and the value at the accessed address.

Function events contain the event type (entry or exit into the function) and the function name.

Depending on the configuration, log records can be produced in text or binary format. In text format, each field of the entry is printed to a file, separated by a delimiter character. In binary format, different types of events are contained in a parent `logentry` structure. The `logentry` structure contains a type field that differentiates the payload as either a function, access or allocation event. The payload is a union between the corresponding three log entry types. In the current version of DINAMITE, each log entry takes 48 bytes total.

Log format involves a surprising trade-off between performance and log size, which we evaluate in the next section.

C. Logging libraries

Instrumented programs do not contain any logic for producing log records. Instead, they contain calls to the externally linked logging library. Our implementation contains three different library versions based on log format and output destination: text-to-file, binary-to-file and binary-to-socket.

The effect of log format on log size

In our implementation, each binary log record takes up 48 bytes of storage. Text entries are variable in size and depend on the number of characters needed to encode all the values. The two extremes of an entry size in text format are:

- Minimum: 18 bytes. Each field can be encoded with a single digit, with added single character delimiters.
- Maximum: 77 bytes. Each field has the maximum value for its storage type in binary format.

TABLE I: Log size comparison in 429.mcf

Number of accesses:	~4.5 billion
Binary log size:	205GB
Text log size:	172GB

TABLE II: Cost breakdown of text and binary formats for 429.mcf, per single log entry

Format	Binary	Text
Base	2.33 ns	
Log cost	13.66 ns	
Format cost	15.99 ns	789.95 ns
Total time (no output)	31.98 ns	805.94 ns

The reality is somewhere in-between, as shown in Table I, which compares log sizes in text and binary format for SPEC2006 429.mcf, with 4.5 billion memory accesses. Text format generates smaller logs; log size is important, because real workloads generate hundreds of gigabytes of logs. At the same time, using text format results in a much higher performance overhead (evaluated in the next section). Since we can forego storing large logs by relying on DINAMITE’s streaming model we always use DINAMITE with the binary log format to avoid the overhead.

The effect of log format on performance

For a detailed insight into the overhead of running the instrumented binary, we break it down into the following components:

- *base cost*: the cost of executing the uninstrumented code
- *log cost*: the cost of invoking the logging library function
- *format cost*: the cost of preparing the log entry for writing
- *output cost*: the cost of writing the log entry

Table II shows the broken-down cost of the instrumentation for 429.mcf. We report all costs except the output cost, because it depends on where we write the log data; the output costs for different output destinations are reported later in this section.

Log cost is fixed and must be invoked for every instrumented event. The only way to reduce this cost is to avoid instrumenting certain events altogether, according to a user-defined criterion at compile time. For example, if we are not interested in exploring the entire memory access trace of a program, but only accesses to a single type, we can tell the compiler to instrument only those. Similarly, we can limit instrumentation to certain functions. Instrumenting isolated data structures or types can discover some memory access patterns, but this kind of filtering is not appropriate for understanding whole program memory behavior or for any analysis involving cache simulation.

Format cost is the cost of packing the log data according to the specified format. It is dominant for the text format, because formatting strings is a very expensive operation, relative to producing a binary record. Text format suffers a 25 \times higher run time relative to the binary format. That is why we always resort to using the binary format in our experiments, despite its higher storage overhead.

Output cost is the cost of writing the log records into a file or sending them over a socket. Besides the cost of accessing the storage medium, it requires a system call. We mitigate

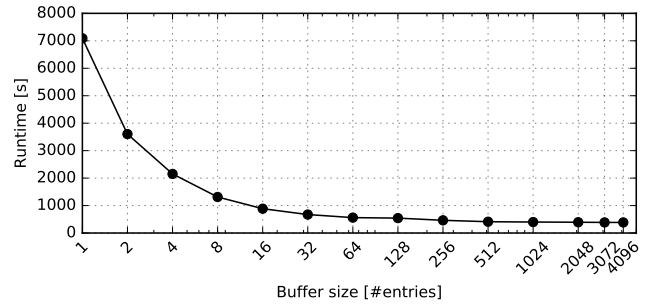


Fig. 2: Impact of buffering on performance of 429.mcf

this overhead via buffering. Figure 2 shows the effects of different buffer sizes on the runtime of 429.mcf in the binary format that writes to a RAM disk. By increasing the output buffer size, performance improvement reaches its maximum at around 20 \times over the unbuffered version. In the rest of our measurements, we used the output buffer of 4096 entries.

Table III compares the performance of 429.mcf instrumented with DINAMITE against slowdowns of two major instrumentation frameworks: Intel Pin and Valgrind. Valgrind performance degradation reported here is obtained from Nethercote et al. [13], and refers to Valgrind’s MemCheck tool which performs memory error checking with a summary output at the end of execution. This is not a fair comparison to access instrumentation, but to the best of our knowledge, a Valgrind tool comparable in functionality with DINAMITE is not available. Numbers for Pin were obtained from the supplied pinatrace tool, output to RAM disk. In Table III, the slowest version of DINAMITE without analysis instruments each access with full debug information (as described in section II-B) and outputs it to a RAM disk filesystem in binary format. Even at this level of detail and with full output enabled, DINAMITE is only 60% slower than Valgrind’s MemCheck and almost 10 \times faster than the comparable access instrumentation in Pin. Even when using the Spark analysis pipeline, DINAMITE is only 35% slower than pinatrace.

Table IV compares the running times for executing 429.mcf with different variants of log formats and outputs. Note that the text-formatted output makes the instrumentation run very slow: 33 \times slower than using the binary format. Sending the trace over a TCP socket to netcat is faster than writing it to a RAM disk. However, introducing Spark Streaming into the pipeline makes the TCP streaming execution 15 \times slower. Optimizing this would require a detailed analysis of Spark’s data receiving system and is left for future work.

TABLE III: Instrumentation overhead comparison - 429.mcf

Framework	Slowdown
Pin (pinatrace output to RAM disk)	354x
Valgrind (MemCheck)	22x
DINAMITE (empty instrumentation)	7x
DINAMITE (binary format, no output)	14x
DINAMITE (binary format, output to RAM disk)	36x
DINAMITE (Spark analysis)	537x

Our design decouples the generation of log records from their processing. Alternatively, embedding analysis logic into

the logging library is also possible. We opted against it for the following reasons:

- Instrumented programs share heap with the logging library. Adding significant bookkeeping data structures to the heap could affect the placement of the program's data and diminish the accuracy of traces.
- Decoupling analysis from logging allows for flexibility in the languages and frameworks used for analyzing memory traces

TABLE IV: Logging library performance - 429.mcf

Version	Destination	Time [s]	Slowdown
Uninstrumented	nil	10.05	1x
Text (unbuffered)	RAM disk	11820	1176x
Binary (file) (buff.)	RAM disk	360.09	36x
Binary (file) (buff.)	Hard disk	1426	142x
TCP (buffered)	netcat >/dev/null	339.12	34x
TCP (buffered)	Spark (access count)	5400	537x

D. Analysis toolkit

The analysis toolkit consists of two different frameworks for writing log processing applications. Logs recorded to a filesystem are processed with the native analysis framework written in C++. The framework includes support for parsing logs and allows the user to easily extend the analysis by writing a new C++ class. Alternatively, the users could write their own parsing and analysis tools in any language of choice. Logs streamed over a TCP socket are processed live with Spark Streaming drivers. Similarly, the user could configure DINAMITE to use any another system to ingest or analyze streaming logs (e.g., Kafka, Google Dataflow). Our toolkit includes a simple cache simulator program, which processes and annotates streamed log records with cache hit/miss indicators. Next, we describe these components in more detail.

1) *Native analysis framework*: The C++ framework provides support for writing arbitrary analysis kernels. To write a new kernel, the programmer must extend the `TracePlugin` class (shown in listing 1).

Listing 1: Trace plugin base class

```

1 class TracePlugin {
2   ...
3   protected:
4     NameMaps *nmaps;
5     TracePlugin(const char *name);
6   public:
7     virtual void processLog(logentry *log) =0;
8     virtual void finalize() =0;
9     virtual void passArgs(char *args) =0;
10 };

```

The framework reads log records into a buffer and passes each log entry to the chosen plugin by invoking its `processLog(logentry*)` method. At the end of the log file, the framework calls the plugin's `finalize()` method, which is used for writing the output of the analysis.

2) *Spark Streaming analysis framework*: To analyze streamed log records with Spark Streaming, the programmer must write an analysis kernel in Scala. To this end, our framework provides a custom Spark Streaming `Receiver` class and a log converter. A receiver accepts batches of log events in binary format over a TCP socket and stores each separate log entry in its associated `StreamingContext`.

Listing 2 shows a Spark Streaming kernel for counting the number of memory accesses per variable. To get useful information out of the entries, the incoming `DStream` is routed through a map operation which invokes our `LogConverter` class on each separate entry. `LogConverter` unpacks and outputs log data as Scala classes, with the distinction between function events, allocation events and access events. To get a `DStream` of instances of a certain event type, logs are filtered with a class matching operation. These events are then mapped to `(varId, 1)` pairs, and reduced by summing over variable IDs. Persistent state is updated by invoking Spark Streaming's `updateStateByKey()` operation. A custom update function, omitted in our listing for brevity, updates the counts by summing new results with the previous state. Results are then output to the console or the filesystem.

Listing 2: Example Spark Streaming kernel

```

1 def main(args: Array[String]) {
2   val sparkConf = new SparkConf()
3   .setAppName("AccessCounter");
4   val ssc = new StreamingContext(sparkConf,
5     new Duration(1000));
6   ssc.checkpoint("/checkpoints/");
7
8   val logs = ssc
9     .receiverStream(new LogReceiver(9999))
10    .map(rawlog =>
11      LogEntryReader.extractEntry(rawlog));
12
13   val counts = logs
14     .filter(log =>
15       log.isInstanceOf[AccessLog])
16     .map(access =>
17       (access.as(...)[AccessLog].varId, 1L))
18     .reduceByKey(_+_ )
19     .updateStateByKey(sumUpdater);
20
21   counts.print();
22
23   ssc.start();
24   ssc.awaitTermination();
25 }

```

Integration with Spark Streaming gives the programmer access to the full set of Spark Streaming operations and can process logs as they are output from a live running program.

3) *Cache simulator*: For detailed analysis of program cache behaviour, we wrote a simple cache simulator, which is placed as an intermediate step between the generation of the log output and the analysis framework (or the filesystem, if we are saving the logs for offline processing).

We simulate a single-level cache, typically configured with parameters reflecting a last-level cache on our target system. The cache simulator accepts log entries over a socket, much like the Spark Streaming analysis framework. It annotates each

TABLE V: 429.mcf top miss offenders

Variable Name	File	Line	Miss count
arc.ident	pbeampp.c	167	45247271
node.orientation	mcfutil.c	85	1104784
node.basic_arc	mcfutil.c	86	988543
arc.cost	mcfutil.c	86	767273
node.potential	pbeampp.c	170	235696

memory access with an indicator whether this was a cache hit or a miss. The annotated logs are passed on to either the analysis framework, or stored in a filesystem for offline processing.

In our evaluation of the system, we found that having cache behaviour information was essential for identifying certain optimization opportunities.

III. EVALUATION

In this section we describe three tools that we built using DINAMITE and demonstrate how they guided our optimization of three applications: SPEC’s *429.mcf*, PARSEC’s *fluidanimate* and *WiredTiger*, a MongoDB’s key-value store.

All experiments described in this section were performed on machines listed in Appendix A.

A. Identifying cache offenders

”For large working sets it is important to use the available cache as well as possible. To achieve this, it might be necessary to rearrange data structures. While it is easier for the programmer to put all the data which conceptually belongs together in the same data structure, this might not be the best approach for maximum performance.”

Ulrich Drepper, 2007[21]

This tool identifies program variables and source lines that generated the most last-level cache accesses and misses. It works as follows:

429.mcf

429.mcf performs single-depot vehicle scheduling using the network simplex method. The implementation represents nodes and arcs in the network as C structs. In the benchmark description the author mentions reordering fields of both node and arc structs in an attempt to reduce cache misses and improve performance [22]. Nevertheless, DINAMITE enabled additional optimizations.

Table V shows the output of the cache-offender tool. We notice that a disproportionate number of misses are being caused by the `ident` field of the `arc` struct, more than four times as many as the second most accessed field, `node.orientation`.

Upon closer inspection, we noticed that all the `arc` structures are allocated as a single large array, even though they represent nodes in a linked data structure. The majority of accesses to `arc.ident` were made within a single loop (shown in Listing 3). The loop iterates over the `arc` array until it finds a match, and only then accesses its other fields.

Every time `arc.ident` was accessed, the corresponding cache line was filled with other fields, most of which were not used before the cache line was evicted. These data layout and access pattern waste cache space and memory bandwidth. We addressed the problem by restructuring the array of `arcs` from the *array of structures* layout into the *structure of arrays*.

Listing 3: 429.mcf pbeampp.c excerpt

```

165 for( ; arc < stop_arcs; arc += nr_group )
166 {
167     if( arc->ident > BASIC )
168     {
169         /*red_cost = bea_compute_red_cost(
170             arc);*/
171         red_cost = arc->cost - arc->tail->
172             potential + arc->head->
173             potential;
174         if( bea_is_dual_infeasible( arc,
175             red_cost ) )
176         {
177             basket_size++;
178             perm[basket_size]->a = arc;
179             perm[basket_size]->cost =
180                 red_cost;
181             perm[basket_size]->abs_cost =
182                 ABS(red_cost);
183         }
184     }
185 }

```

Our modifications brought a 55% reduction in LLC misses, and a 12% improvement in the overall runtime.

fluidanimate

Fluidanimate is an Intel *Recognition, Mining and Synthesis* application that uses the Smoothed Particle Hydrodynamics method to simulate an incompressible fluid. It uses the Navier-Stokes equation to derive fluid density fields. It is included in the PARSEC 3.0 benchmark suite because of the increasing significance of physics simulation in video-game programming and real-time animation domains [23].

Profiling *fluidanimate* with `perf` [10] showed that it has a high LLC miss rate of 30% on our system. We instrumented the program using DINAMITE and ran it through the cache-offender tool. Table VI shows output of the tool: variable names and source lines responsible for the most cache misses. The top cache offenders are `Cell.next` and `Vec3.x`. The names of the structs and fields suggest that a `Cell` is an element of a linked collection. Listing 4 shows the code excerpt pointed to by the output of our tool. We can immediately see that the code generating misses is a traversal of grid of `Cell` structures in which only the `next` field is touched.

Looking at the definitions for `Cell` and `Vec3` types we can see that `Cell` represents a linked list of containers for arrays of `Vec3` structures that contain three-dimensional vectors. The arrays themselves are contained within the `Cell` struct in their entirety. The total size of a `Cell` struct with the payload was 896 bytes, making a single instance span 14 cache lines.

This data layout is poorly optimized for traversing lists of `Cells`, because each new `Cell` access generates a cache miss. Our idea, therefore, was to allocate the `Cell`’s payload,

TABLE VI: CSV output of the miss summary tool for fluidanimate

Variable name	File	Line	Miss count
Cell.next	pthread.cpp	530	184496
Vec3.x	./fluid.hpp	354	95682
Cell.next	./fluid.hpp	404	73800
Vec3.x	./fluid.hpp	346	67327
Vec3.x	./fluid.hpp	355	66657

which is rarely touched, separately from the rest of the structure. The structure would then include a pointer to its payload; since each `Cell`'s payload consists of multiple arrays, adding a layer of indirection to access the payload would not be much of a penalty. Allocating the `Cell` payload separately brings the size of the structure down to 16 bytes. Since consecutive calls to a memory allocator function for a variable of the same size will return near consecutive addresses in most standard libraries, consecutive `Cells` will be allocated close together, and several of them will fit into a single cache line.

Listing 4: fluidanimate pthreads.cpp code excerpt

```

522 void ClearParticlesMT(int tid)
523 {
524     for(int iz = grids[tid].sz; iz < grids[
        tid].ez; ++iz)
525         for(int iy = grids[tid].sy; iy < grids[
            tid].ey; ++iy)
526             for(int ix = grids[tid].sx; ix <
                grids[tid].ex; ++ix)
527             {
528                 int index = (iz*ny + iy)*nx + ix;
529                 cnumPars[index] = 0;
530                 cells[index].next = NULL;
531                 last_cells[index] = &cells[index];
532             }
533 }

```

This change brought a 50% reduction in the LLC cache miss rate and a 15% reduction in runtime with 16 threads (see Figure 3).

An interesting observation is that `Cells` were allocated in the original implementation to be cache-aligned and padded to a fill the entire cache line, indicating a prior effort to make better use of the cache hierarchy. However, with DINAMITE we discovered that making the `Cell` struct larger by padding actually hurt performance on our system. Intricacies of modern multi-core memory hierarchies can mislead even very experienced programmers. Powerful performance analysis tools are thus crucially important.

B. Structure splitting

Our structure splitting tool is based on the class splitting algorithm proposed by Chilimbi et al. for Java programs [17]. The algorithm analyses how the program accesses the members of a class to determine if a class is fit to split into two separate classes. Splitting classes is motivated by the idea that *hot* fields, or fields that are accessed significantly more than *cold* fields, should be placed in a separate class so that more hot data can be packed into a single cache block. To access fields that are considered cold, the hot class includes a pointer

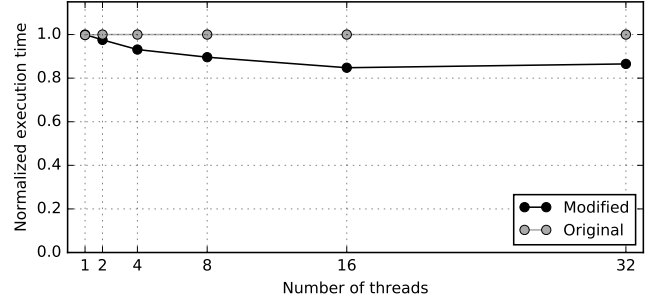


Fig. 3: Scaling improvements in PARSEC3.0 fluidanimate application

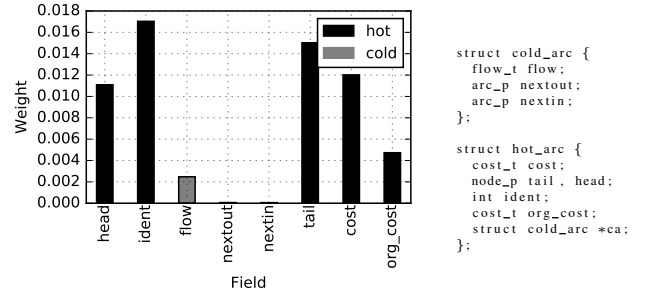


Fig. 4: Tool output and modified code for structure splitting of 429.mcf

to the cold class. (This is similar to the optimization that we applied to *fluidanimate*).

The algorithm begins by identifying *live* classes. A class is considered *live* if it is accessed more than a certain threshold, and only live classes are considered for splitting. Next, fields in live classes are marked as hot or cold depending on how many times their respective class is accessed. If a field is accessed significantly more than other fields, it is considered hot. Full details of the algorithm are described elsewhere [17].

Chilimbi et al. implemented the splitting algorithm for Java classes, using the JVM for access statistics and a Java byte-code instrumentation tool BIT [24]. We implemented the splitting algorithm in DINAMITE, making it accessible to a wider range of programs, including those written in unmanaged languages. The tool works as follows:

The Spark Streaming driver receives access logs from the instrumented binary and produces the list of variables and corresponding access counts. Then a Python script generates a chart for each live structure showing weights assigned by the algorithm for individual fields; black bars for hot fields and gray bars for cold fields. Programmers then split their structures according to the hot and cold fields in the chart.

Figure 4 shows the chart produced by the structure splitting tool for the `arc` struct in *429.mcf* as well as the modified `arc` struct code. Similarly struct `node` (not shown) was another live struct with both hot and cold fields. Splitting hot and cold fields in these structs delivered 20% speedup and reduced the LLC miss rate by 60%, as measured with `perf`.

C. Shared variable detection

On machines with even a handful of cores, variables updated by multiple threads can quickly become a scaling bottleneck [6], even if these variables are not protected by a lock or accessed via atomic instructions [25]. Repeatedly updating a shared variable from different cores stresses the coherency protocol and can slow down the program by an order of magnitude relative to a sharing-free execution. Tools for detecting shared variable bottlenecks do exist, but they are hardware-specific (e.g., Intel’s VTune [26] works only on Intel machines, DProf [5] and Memprof [4] work only on AMD hardware) and can be non-trivial to set up (DProf and Memprof require changing the kernel). DINAMITE is easily extended to detect shared variable bottlenecks on any binary that can be compiled with LLVM.

To demonstrate, we created a simple tool that we then used to find a known scalability bottleneck in WiredTiger [18] [20], a MongoDB storage engine [19]. To truly test the experience of creating new tools for DINAMITE, the student who created the shared variable detection tool was *not* informed what variables and source locations triggered the bottleneck; he was only advised that the bottleneck exists and provided the instructions for running the problematic workload.

The engineer who originally diagnosed the scalability issue took about week to do so after observing poor performance; she used Memprof, which required communication with its authors and changes to the kernel. Even though the changes were simple, they would likely be considered “beyond the call of duty” by many developers. The DINAMITE tool took several hours to create by a person familiar with the overall framework and consisted of two simple Spark Streaming kernels and a Python script.

The first and second kernel identify top shared variables. The second kernel processes the execution traces again, looking only for frequently shared variables and collects the source locations where the accesses are made. The tool could be structured with only a single kernel that both identifies the top shared variables and records the source lines, but we found that having two kernels is simpler and results in better performance of Spark Streaming.

The first kernel translates memory access log entries into (*accessed address*, *variable identifier* and *thread identifier*) tuples. Each tuple acts as the key in *map-reduce* transformation that produces a list of variable-identifying tuples and the corresponding total memory accesses. The result is stored in a persistent table.

Next, a Python script reads that table and transforms the data into a dictionary, where each accessed address serves as a key and the corresponding value contains the variable identifier and access counts performed by each thread. The script filters the results according to the following criteria:

- It removes all entries accessed by only a single thread
- It removes all entries that are not uniformly shared by threads. We define uniform sharing as follows:

Let A_{sorted} be a list of all the per-thread access counts

TABLE VII: Most accessed shared variables

Address	# Accesses	# Threads	Variable
0x64D900	42495568	32	__wt_stats.v
0x64D1A4	26183326	74	__wt_connection_impl
0x64E0EC	7233836	72	__wt_connection_impl.N/A
0x64D100	4786616	36	__wt_txn_global.states
0x64D540	4786370	34	__wt_stats.v

for the address, sorted in descending order, zero indexed. if $A_{sorted}[0] < 2 * A_{sorted}[1]$ the sharing is uniform.

The output is then sorted in descending order by the total number of accesses. Table VII shows the first five entries of the output generated for the LevelDB sequential read benchmark over WiredTiger (release 2.6.0) executed with 32 threads¹, which triggered the bottleneck. The top offender is the field *v* in *__wt_stats* struct.

These results only point to the variable responsible for shared accesses. To find the root cause, we use the second Spark kernel to find the source location where the accesses are performed. The second kernel is very similar to the first, except it discards all the log entries that do not correspond to the top shared variable, and keeps the source lines where the accesses occurred. We sort the results by the number of accesses in descending order.

Listing 5: WiredTiger shared variable analysis result (JSON)

```

1  {
2    "threadcount": 18,
3    "totalcount": 311881,
4    "threads": [
5      [
6        156,
7        19492
8      ],
9      [
10       163,
11       19520
12     ],
13     ...
14   ],
15   "file": "wiredtiger/build_posix/./src/
        btree/bt_curnext.c",
16   "line": 446,
17   "variable": "__wt_stats.v"
18 }

```

Listing 5 shows the first entry of the output (redacted for brevity), which correctly identifies the source location responsible for the bottleneck. The fields in the output JSON document are self-explanatory apart from the *threads* list, which contains (*thread id*, *thread access count*) pairs. It turns out that this sequential read-only workload suffered from scalability problems, because threads were incrementing a shared statistics counter after each read operation. This problem was later fixed by implementing per-thread statistic buffers.

Figure 5 shows the performance impact of the bug on Machine A (described in the appendix). A seemingly benign

¹The connection structure is shown as accessed by more than 70 threads because the benchmark creates and tears down additional threads before the measured run.

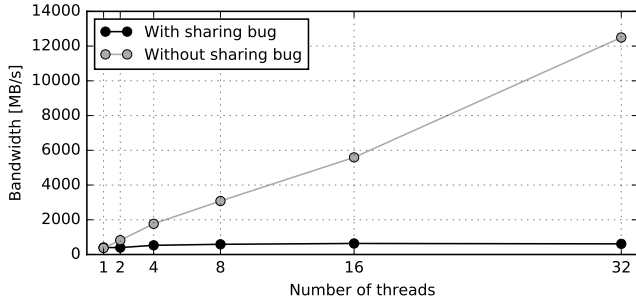


Fig. 5: Scaling improvements WiredTiger after removing the shared variable bug

counter increment, which takes negligible time in a single-threaded execution, quickly escalates into a huge scaling bottleneck with as few as four threads and slows down the workload by an impressive factor of 20 with 32 threads. Previous work reported similar performance impact of shared variables on multicore machines [25]. With the increasing core counts on new hardware the importance of tools that enable productive memory performance analysis will continue to grow.

IV. RELATED WORK

Several instrumentation frameworks have been designed previously with the goal of solving memory bottleneck problems. Limitations of these frameworks include targeting a specific programming language, providing coarse grained instrumentation abstractions, or providing fine grained instrumentation abstractions that are sampled to reduce overhead. Other tools are not as flexible in that users can only view output through a user interface, or are designed for a specific use case.

Existing instrumentation frameworks like Pin [14] and Valgrind [13] instrument programs and allow programmers to build dynamic analysis tools on top of them much like DINAMITE. Pin instruments using a highly optimized JIT compiler that intercepts the first instruction of a supplied executable and compiles instrumentation functions to execute where appropriate. Valgrind recompiles code blocks from the binary into its own IR, instruments them using a tool plugin, and compiles them back to machine code to be executed. Additionally, Valgrind provides a fast shadow memory implementation for use in the tool plugins. Both of these frameworks have access only to information contained within the executable, making it difficult to relate machine instructions to source-level context. DINAMITE instruments programs at compile time, at which point all the source-level information is available and automatically passed to the logging functions. The need for shadow memory is eliminated by decoupling DINAMITE’s instrumentation from its analysis frameworks.

Zhao et al. [27] describe a tool designed to detect true and false sharing built on top of the memory shadowing framework Umbra [28]. Memory sharing is detected by associating cache lines with shadow memory exposed by Umbra. Association of thread to address is done via a bitmap describing thread ids

responsible for each access. Sheriff [29] addresses the same problem, but requires either the programmer to rewrite source code or rely on sampling to catch culprits, and is specifically designed to detect false sharing. DINAMITE achieves the same result by inserting thread IDs in each log entry, which also contain accessed addresses. The log entry also contains all the source level details necessary to pinpoint exactly where in the code the accesses were performed and to what data type.

Memprof [4] is a tool that profiles objects that make remote memory accesses on NUMA machines so programmers can potentially minimize them. Memory accesses are measured through instruction-based sampling (IBS) that relies on hardware support and requires using a kernel module, constraining the tool to the Linux/AMD platforms. DINAMITE can be extended with a native plugin or Spark kernel to generate the same information. It sacrifices performance over accuracy and portability as it does not rely on hardware support for instruction sampling.

Similar to Memprof, DProf [5] uses IBS to acquire memory traces to locate data types that stress the cache. Programmers can view data type statistics and how they behave in the cache, what data types generate the most cache accesses and misses, and the most common functions that operate on these data types. DProf required changes to the operating system, which is significant barrier for its adoption. The flexibility of DINAMITE enabled us to plug in a cache simulator into the framework and to generate the same information as DProf, but with greater flexibility to add new analysis and without attachment to a particular operating system or hardware.

Other tools provide more source-level detail but are slower and less flexible. Memspy [30] provides rich information on program execution, including the execution time, miss rate, and memory stall time broken down by code and data. Details are tracked by executing the application through a memory simulator and instrumentation through a preprocessor, which is not as portable as LLVM. Memspy reported a $22\times$ - $58\times$ slowdown in execution time. DINAMITE’s slowdowns are competitive with modern instrumentation frameworks and provide a pluggable framework for all kinds of data analyses.

Like MACPO [31], our tool instruments data accesses at the compiler level instead of the binary level to keep source-level information. To combat overhead, MACPO limits instrumentation to “snapshots” of program execution, that are staggered in an attempt to capture complete program behaviour. Trace size is reduced by limiting instrumentation not only to “snapshots” but also to non-scalar data types. Similarly Dprof and Memprof use IBS and only output the most commonly accessed data and their cache statistics. DINAMITE instruments all memory accesses inviting unlimited flexibility of analysis at the expense of higher runtime overhead.

V. FUTURE WORK AND CONCLUSIONS

We described the implementation of DINAMITE and discussed the performance implications of our design choices: a fine-grained compiler-based instrumentation and a flexible analysis framework. Our detailed breakdown of the costs

involved in doing instrumentation of memory accesses leads us to conclude that this kind of design is not only feasible, but allows for a great level of detail in the generated traces, while keeping the slowdown comparable to the state-of-the-art instrumentation frameworks. We introduced a novel fusion of instrumentation and stream processing that eliminates the need for storing traces and provides an easy to use Spark Streaming API for analysis purposes. Finally we demonstrated the utility of DINAMITE by performing three types of analysis that were difficult, impossible, or constrained to a certain OS/hardware platform with the previously available tools.

In future work, we plan to expand on the kinds of analysis that can be done on access traces using a streaming framework. Spark Streaming currently buffers incoming log records and processes them at the expiration of a configurable timeslice. In our experience, the timeslice must be rather large (e.g., one second) to avoid performance problems with Spark, but with such a large timeslice the batches contain hundreds of thousands of accesses. Adding support for a framework that is able to process a batch of records after it accumulates a specified *number* of records would let us have finer granularity in our analysis. This kind of setup would allow discovering access patterns within small windows of time, which is important for certain optimizations.

Further, we plan to explore and optimize the slowdown of DINAMITE when using the full analysis pipeline with Spark Streaming. Finding the best way to integrate the log generation and analysis is an important factor in improving the overall productivity of engineers using our system.

Finally, our implementation of the cache simulator is rather simplistic. Adding a multi-level cache simulator such as Dinero IV [32] and adding more cache information to the logs would help improve understanding how different data organization and access patterns affect program efficiency.

APPENDIX HARDWARE

All hardware in our evaluation was performed on one of the following machines:

- *Machine A* - AMD Opteron 6272, 4 chips with 16 cores and 16MB of last level cache each, and 512GB of RAM
- *Machine B* - AMD Opteron 2435, 2 chips with 6 cores 6MB of last level cache each, and 32GB of RAM

REFERENCES

- [1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "Dbmss on a modern processor: Where does time go?" in *Vldb'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, no. DIAS-CONF-1999-001.
- [2] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafae, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ACM SIGPLAN Notices*, vol. 47, no. 4.
- [3] C. Lattner and V. Adve, "Automatic pool allocation: improving performance by controlling data structure layout in the heap," in *ACM SIGPLAN Notices*, vol. 40, no. 6.
- [4] R. Lachaize, B. Lepers, and V. Quéma, "Memprof: A memory profiler for numa multicore systems," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*.
- [5] A. Pesterev, N. Zeldovich, and R. T. Morris, "Locating cache performance bottlenecks using data profiling," in *Proceedings of the 5th European conference on Computer systems*.
- [6] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, N. Zeldovich *et al.*, "An analysis of linux scalability to many cores," in *OSDI*, vol. 10, no. 13.
- [7] K. Goto and R. A. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3.
- [8] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha, "Cache-oblivious mesh layouts," in *ACM Transactions on Graphics (TOG)*, vol. 24, no. 3.
- [9] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*.
- [10] A. C. de Melo, "The new linuxperf tools," 2010.
- [11] G. Bitzes and A. Nowak, "The overhead of profiling using pmu hardware counters," *CERN openlab report*, 2014.
- [12] M. Itzkowitz, B. J. Wylie, C. Aoki, and N. Kosche, "Memory profiling using hardware counters," in *Supercomputing, 2003 ACM/IEEE Conference*.
- [13] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6.
- [14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM sigplan notices*, vol. 40, no. 6.
- [15] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*.
- [16] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Presented as part of the*, 2012.
- [17] T. M. Chilimbi, B. Davidson, and J. R. Larus, "Cache-conscious structure definition," in *ACM SIGPLAN Notices*, vol. 34, no. 5.
- [18] (2016) Wired tiger: making big data roar. [Online]. Available: <http://www.wiredtiger.com/>
- [19] (2016) Wiredtiger storage engine. [Online]. Available: <https://docs.mongodb.com/manual/core/wiredtiger/>
- [20] (2016) Wt-2029 improve scalability of statistics. [Online]. Available: <https://github.com/wiredtiger/wiredtiger/pull/2102>
- [21] U. Drepper, "What every programmer should know about memory," *Red Hat, Inc*, vol. 11.
- [22] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, 2006.
- [23] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.
- [24] H. B. Lee and B. G. Zorn, "Bit: A tool for instrumenting java bytecodes," in *USENIX Symposium on Internet technologies and Systems*.
- [25] D. Dice, Y. Lev, and M. Moir, "Scalable statistics counters," in *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*.
- [26] R. K. Malladi, "Using intel® vtune performance analyzer events/ratios & optimizing applications," <http://software.intel.com>, 2009.
- [27] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe, "Dynamic cache contention detection in multi-threaded applications," in *ACM SIGPLAN Notices*, vol. 46, no. 7.
- [28] Q. Zhao, D. Bruening, and S. Amarasinghe, "Umbra: Efficient and scalable memory shadowing," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*.
- [29] T. Liu and E. D. Berger, "Sheriff: precise detection and automatic mitigation of false sharing," *ACM SIGPLAN Notices*, vol. 46, no. 10.
- [30] M. Martonosi, A. Gupta, and T. Anderson, "Memspy: Analyzing memory system bottlenecks in programs," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 20, no. 1.
- [31] A. Rane and J. Browne, "Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*.
- [32] J. Edler and M. D. Hill, "Dinero iv trace-driven uniprocessor cache simulator," 1998.